

The FSM Network Model for Behavioral Synthesis of Control-Dominated Machines

Wayne Wolf
Department of Electrical Engineering
Princeton University

Abstract

Since many ASICs are dominated by control functions, control-dominated architectures form an important domain for behavioral synthesis. We propose modeling control-dominated architectures during behavioral synthesis as networks of communicating FSMs—the model more directly reflects behavior and allows more accurate cost estimation, especially for control, than do traditional data-directed representations for control-dominated machines. We show how to implement a number of important compiler optimizations on the FSM network model.

1 Introduction

Many ASICs are *control-dominated*—most of their logic devoted to FSMs and random logic rather than adders and other architectural-sized primitives [1]. Because of the large volume of ASIC design and the time pressure under which most ASICs are designed, ASICs—and, by implication, control-dominated architectures—represent an important target for behavioral synthesis.

A great deal of work has been done on behavioral synthesis, but most work has concentrated on datapath-controller design [2] [3] [4] [5], digital signal processor design [6], or interfaces [7] [8] [9]. We believe that new methods are required for control-dominated architectures.

An important aspect of a synthesis methodology is design representation. Most work in behavioral synthesis has been based on control-data flow graph models [10] [11] [12]. The thesis of this paper is that networks of communicating finite-state machines are an effective, efficient model for behavioral synthesis of control-dominated machines. An FSM network model for a machine partitions the design into finite-state machines and combinational units. Control-dominated parts of the design are placed in FSMs (and possibly adjunct combinational blocks); data registers can be modeled as separate FSMs, either explicitly as state transition tables or implicitly as black boxes; special function units can also be placed in separate partitions.

The advantages of the FSM network model include:

- The model supports both behavioral and structural optimization.
- Compilation algorithms can blur the distinction between data and control.
- Many algorithms can be designed to use the FSM network as both input and output. Such algorithms can be made more specialized because they allow more freedom in how they are interconnected to optimize for multiple criteria.
- Because the model is based on Boolean algebra and automata theory, it has well-defined notions of composition, decomposition, and minimality of representation.
- Implementation costs, particularly the cost of control, can be cleanly measured—simple estimates can be made quickly, while very accurate estimates can be made with more CPU time.

Any design representation for behavioral synthesis must be able to support a number of optimizations that have been shown in programming languages [13] and behavioral synthesis [14] [15] [16] to be important.

The bulk of this paper is devoted to demonstrating how to implement a variety of optimizations on the FSM network model. This paper is not intended to present a complete compilation methodology, but only to show the utility of the FSM network model for solving problems peculiar to control-dominated architectures. We have built a preliminary translation and optimization system to experiment with FSM network optimizations; we will illustrate techniques using example designs from the Behavioral Synthesis Workshop benchmarks and elsewhere.

Research at UC Berkeley [17] and Stanford [12] is pursuing behavioral synthesis of control-dominated architectures, both of these efforts use dataflow-based models for system behavior. The FSM network model is not equally suited to all target architectures—machines with a great deal of data and relatively little control, like DSPs and datapaths, are poorly modeled by FSM networks. Our experimental compiler shows, however, that it is an effective representation for control-dominated machines.

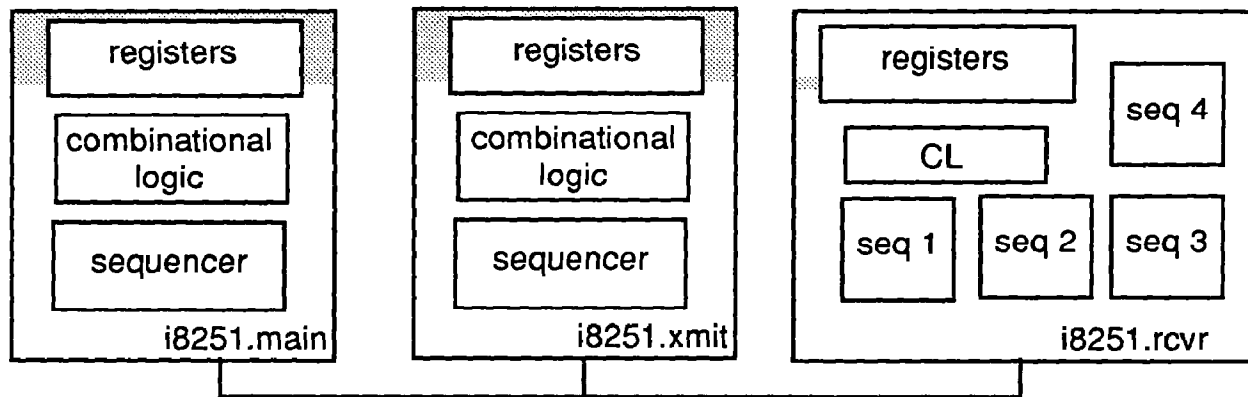


Figure 1: The translated structure of the i8251 benchmark.

2 The compilation process

We derive our *translation + optimization* methodology for behavioral synthesis from programming language compilation. The source program is translated into a naive intermediate implementation; the intermediate form is optimized to meet the user's area and delay requirements. As in programming language compilation, there may be many translation + optimization steps: some optimizations, like constant folding, are best done at the program level [18]; behavioral synthesis concentrates on cost-directed optimizations which create and modify the register-transfer design; the completed RT design is optimized by sequential [19] [20], logic [21], and layout algorithms. Our research concentrates on optimization below the program level and above the register-transfer level.

The first step in compilation is to produce an FSM network which represents the design. Our current compiler directly maps the structure of the behavior description into the initial FSM network: one FSM is built per block in the behavior description; registers are implemented as separate blocks. Statements in the behavior program map into one or more state and transition. Figure 1 shows the network structure of the i8251 USART: the chip is partitioned into three subsystems, the largest of which is divided into four FSMs.

The behavior description of each component must be constructed from the program block which defines its behavior. Figure 2 shows how to build a state transition graph fragment for a code fragment. Syntax-directed translation followed by optimization is a good compilation strategy in behavioral synthesis for the same reasons as in programming languages: it reduces the opportunity for introducing bugs into the translator; and it defers global decisions about optimization until the information needed to make them is available.

One key performance constraint in ASIC design is

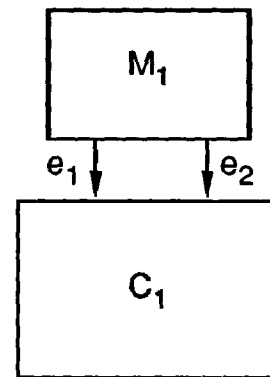


Figure 3: A behavior constraint machine.

clock cycle time, which is easy to specify and estimate in the FSM network model. Sequential behavior constraints can be specified as constraint machines; a similar method was used by Clarke, *et al.* to describe verification conditions [22]. In Figure 3, events e_1 and e_2 in M_1 are required to happen three clock cycles apart. The phantom outputs e_1 and e_2 signal when the events occur in M_1 ; the constraint machine C_1 measures the number of cycles between these events. C_1 can be collapsed into M_1 , with C_1 's states used to mark the relevant states in M_1 for checking and correction. Since the constraint machine can be an arbitrary FSM, more complex interface conditions can also be expressed.

Optimization moves the FSM network design from an initial implementation, which executes the desired behavior but does not meet the user's area and delay requirements, into a different implementation which does meet the user's constraints. A number of FSM networks execute the program's behavior, but not all at the same cost. We want to improve the design not only by changing the FSM implementation (state assignment, logic, layout)

```

wait (a || b);
if c then
  x = e
else
  x = a && d;

```

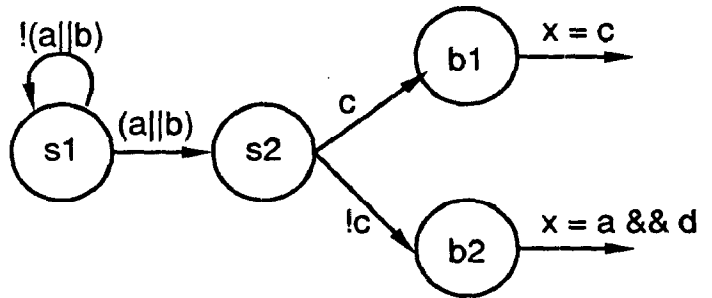


Figure 2: Syntax-directed translation of behavior into FSMs.

but also by changing the FSM network's I/O behavior—the cycle-by-cycle behavior of the FSM network—within the bounds imposed by the behavior program's behavior description.

We divide FSM network optimizations into two categories:

- *Behavioral optimizations* change the terminal behavior of an FSM or an FSM network, within the constraints of the program's behavior.
- *Structural optimizations* may change state transition graphs by changing the partitioning of the system or the representation of a component, but they do not change the terminal behavior of the FSM or FSM network on which they operate.

We will talk about optimizations in both categories, but our current set of optimizations is not complete.

The Yorktown Silicon Compiler [5] showed how combinational logic optimization can take care of many transformations on strictly combinational logic and showed what transformations cannot be subsumed by combinational logic optimization. The optimizations described here are built on top of combinational logic optimization.

3 Structural optimizations

Snow [14], Trickey [15], and Johnson [16] identified a number of important optimizations, some of which we categorize as program-level or functional and some of which are structural. Among the important structural optimizations are:

- loop unrolling;
- inline function expansion;
- register elimination;
- dead code elimination.

These optimizations can be performed using automata theoretic algorithms [23]. All these transformations preserve terminal behavior—the old and new machines cannot be distinguished by observing their input/output ports.

Important automata theoretic algorithms that can be used to implement structural optimizations include:

- *partitioning*, which breaks an FSM into two or more connected FSMs;
- *collapsing*, which combines two or more FSMs into a single FSM by taking the Cartesian product of their states and transitions;
- *minimization*, which finds sets of states that cannot be distinguished by the machine's I/O behavior and replaces them with a single state.

The behavior program written by the user has many syntactic hints on design partitioning. Given the current state of FSM decomposition algorithms, the best compilation strategy seems to be to compile the design into many small FSMs and then collapse them where expedient, rather than compiling the program into one large FSM and then decomposing it. We will concentrate on how collapsing and minimization transform the partitioning of behavior among components.

3.1 Loop unrolling and flag/register elimination

Loop unrolling, inline function expansion, and register/flag elimination are closely related. All are implemented by collapsing and minimization.

Figure 4 shows a loop from the multiqueue example; the code fragment uses an external counter to wait the required n cycles between incoming/outgoing data words. The direct translation of the behavior description results in two modules: a sequencer executing the loop code and a counter with the loop state.

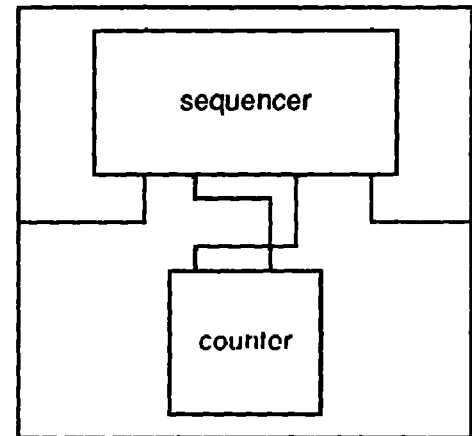
The loop is unrolled by collapsing together the two machines. The simplest way to collapse the two machines is to form the Cartesian product of the states and transitions of the sequencer and counter machines. That naively collapsed machine, however, has many equivalent states [23]—states which cannot be distinguished by any input/output sequence. FSM minimization produces an equivalent machine with the minimum number of states. Collapsing followed by minimization can produce an extremely large intermediate machine even if the minimized machine is small. We have developed a new algorithm to minimize while collapsing [24]; this algorithm generates

```

wait.for.data := BEGIN
! count down clock cycles
! to next data word
waitcounter.reset = 1 NEXT
waitloop := BEGIN
  waitcounter.incr = 1 NEXT
  if (NOT waitcounter.done) =>
    LEAVE waitloop NEXT
  RESTART waitloop
END
END

```

ISPS description



FSM network structure

Figure 4: A loop structure to be unrolled.

small product machines, making feasible the analysis and modification of a much larger class of designs by collapsing.

Eliminating registers and flags (flags are one-bit registers) helps the compiler blur the distinction between control and data. A flag connected to an FSM is used by the FSM as data, but it also represents externally stored state. Collapsing the flag into the FSM which uses it changes the representation of the machine. Eliminating small registers uncovers new optimizations in the machine.

Figure 5 shows how eliminating a flag causes expansion of the code in the absorbing FSM. The test of the flag caused a branch in the original machine; eliminating the flag causes there to be two copies of the pre-branch states and transitions. The post-branch transitions can be separately optimized against the pre-branch transitions in the expanded machine.

Counters are especially amenable to elimination because of the structure of their state transition graphs. For example, a minimal version of a 7-state sequencer collapsed with a 100-state counter produces a 403-state machine, compared with 700 states in the Cartesian product machine; we can perform the collapsing in 17 CPU seconds on a Sun Sparcstation-1.

3.2 Other optimizations

The FSM network analog of inline function expansion is collapsing the FSM implementing one function into the FSM which calls the function. Inline expansion allows behavioral optimizations to change behavior across the boundary of the two original FSMs. For small machines the cost is small. We have a queue processor designed as three components: an enqueue machine of two states, a dequeue machine of two states, and a master controller

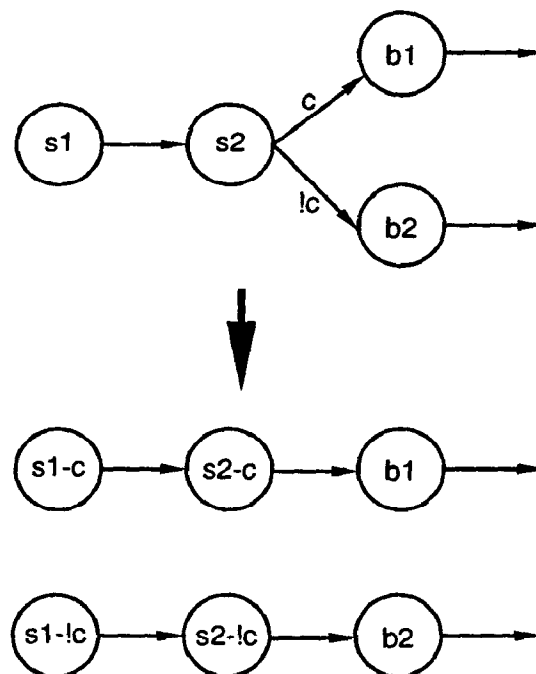


Figure 5: How flag elimination affects the state transition graph.

of 7 states, so the enqueue and dequeue machines can be expanded into the controller at very little cost.

Minimization implements one form of dead code elimination—the elimination of states and associated transitions which do not add to the behavior of the machine.

3.3 Partitioning and minimality

Traditional data-control flow models of behavior are constructed using architectural-sized components such as adders and multiplexers. These data structures do not generally have associated with them complete algebras for manipulating combinations of these primitives: combining a mux with an adder or merging a control branch into a multiplexer, for example. The lack of an algebra on the primitive operations limits the number of ways the design can be repartitioned and prevents us from defining minimality of a behavior description.

The primitives in the FSM network model are the Boolean functions and state transition tables, both of which have well-defined algebras. As a result, we have a great deal of flexibility in repartitioning the behavior description and we can generate a minimal representation of a given partition. This wider range of partitions of behavior allows the compiler to expose new candidate optimizations. Generating a minimal collapsed machine is one example of partitioning with minimality; another is collapsing a combinational logic function (such as a multiplexer) into a FSM and minimizing the representation. Working from a minimal representation of the components has two practical advantages: it saves CPU time by reducing the size of the description; and it makes the results of synthesis more predictable by eliminating artifacts of description in the user's source program.

3.4 Cost estimation

Because the FSM network model can be directly implemented as layout, it is easy to estimate implementation costs which can be used to guide optimization. Rough area estimates for FSMs are easily available by counting the number of states or the a weighted function of the number of states and transitions in the FSMs. Such estimates can be used to steer the compiler away from gross errors.

Much more accurate estimates can be made by encoding the FSMs and implementing them in multi-level logic form. State assignment algorithms are quick [20]. Experiments by Lightner and Wolf [25] show that counting literals in the optimized multi-level network is an accurate estimate of grids (and therefore final chip area) and that simple optimization scripts can produce a logic implementation good enough for area estimates. The multi-level network can also be used to estimate FSM clock cycle time by measuring the depth of the network.

Similar techniques can be used to estimate area and delay for combinational and sequential components not represented as state transition graphs. Logic synthesis can

be used to build an approximate implementation which can be used to estimate costs. Lightner and Wolf [25] also describe how the accuracy of area estimation varies with the CPU time expended on optimizing the structure.

Estimating the area and delay of components does not accurately estimate the area and delay of an arbitrary-block layout composed from those components. McFarland [26] showed that good global estimates are critical to successful behavioral synthesis. If the implementation medium is a standard-cell layout, accurate global estimates can be made by performing logic optimization simultaneously on all the components; otherwise we expect that the floorplanning methods used by Bud would be applicable.

4 Behavioral optimizations

Behavioral optimizations modify the cycle-by-cycle behavior of an FSM or FSM network, while maintaining congruence with the intended behavior. We have developed one behavioral optimization, which we call *state scheduling*, which we briefly describe here to help explain what types of transformations on FSM behavior can be made by behavioral optimizations. Other behavioral optimization algorithms, both for area and delay, will be required for high-performance behavioral synthesis.

State transition graphs generated by syntax-directed translation, such as the one in Figure 2, often have more states than are required achieve the desired behavior. State scheduling reduces the size of an FSM by minimizing the number of states in the machine, thereby roughly minimizing the machine's area. State scheduling differs from most scheduling methods in that it minimizes the cost of the controller rather than the time required for operations.

We can change an FSM's I/O (cycle-by-cycle) behavior while preserving the behavior specified in the source program by properly adding and deleting states and transitions. A state may be required to be added to ensure a minimum distance between two events; adding a state is easy since a null action can be inserted after the new state. When a state is deleted, care must be taken to ensure that all paths through the deleted state are preserved and that all input/output actions along each path are executed.

5 Conclusions

Control-dominated architectures are an important target for behavioral synthesis. The FSM network model is a powerful representation for synthesizing control-dominated machines: it is abstract enough to support useful behavioral optimizations, yet close enough to the implementation of control-dominated machines to allow reasonable cost estimation. We will continue working to develop behavioral synthesis methods based on the FSM network model.

Acknowledgements

The ideas behind this work were developed while the author was with AT&T Bell Laboratories and this work reflects many lessons learned about real-world problems from AT&T designers. Thanks to Kurt Keutzer and Mike Lightner for many hours of discussion about automata theory, FSM networks, and synthesis. Thanks to Miriam Leiser of Cornell for many conversations on moving behavior in FSM networks. Research performed at Princeton was sponsored by the Semiconductor Research Corporation.

References

- [1] Kurt Keutzer. Three competing design methodologies for ASICs: architectural synthesis, logic synthesis and module generation. In *Proceedings, 26th Design Automation Conference*, pages 308–313, ACM/IEEE, June 1989.
- [2] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, and R. L. Blackburn. *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*. Kluwer Academic Publishers, Boston, 1990.
- [3] David Knapp, John Granacki, and Alice Parker. An expert synthesis system. In *Proceedings, ICCAD-83*, pages 164–165, ACM/IEEE, September 1983.
- [4] Wolfgang Rosenstiel and Raul Camposano. Synthesizing circuits from behavioral level specifications. In C. J. Koomey and T. Moto-oka, editors, *Computer Hardware Description Languages and their Applications*, pages 391–403, Elsevier Science Publishers B. V., 1985.
- [5] R. K. Brayton, R. Camposano, G. De Micheli, R. H. J. M. Otten, and J. van Eijndhoven. The Yorktown Silicon Compiler. In Daniel D. Gajski, editor, *Silicon Compilation*, pages 204–310, Addison-Wesley, 1988.
- [6] H. DeMan, J. Rabaey, P. Six, and L. Claesen. Cathedral-II: a silicon compiler for digital signal processing. *IEEE Design & Test*, 3(6):13–25, December 1986.
- [7] John Nestor. *Specification and Synthesis of Digital Systems with Interfaces*. PhD thesis, Carnegie-Mellon University, April 1987. Report No. CMUCAD-87-10.
- [8] Gaetano Borriello. *A New Interface Specification Methodology and its Application to Transducer Synthesis*. PhD thesis, University of California, Berkeley, May 1988. Report No. UCB/CSD 88/430.
- [9] Sally Hayati and Alice Parker. Automatic production of controller specifications from control and timing descriptions. In *Proceedings, 26th Design Automation Conference*, pages 75–80, ACM/IEEE, June 1989.
- [10] Michael C. McFarland, S. J. *The Value Trace: A Database for Automated Digital Design*. Master's thesis, Carnegie-Mellon University, December 1978.
- [11] David W. Knapp and Alice C. Parker. *A Data Structure for VLSI Synthesis and Verification*. Technical Report CRI-85-19, Computer Research Institute, University of Southern California, August 1985.
- [12] Giovanni De Micheli and David C. Ku. HERCULES – a system for high-level synthesis. In *Proceedings, 25th Design Automation Conference*, pages 483–498, ACM/IEEE, June 1988.
- [13] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading MA, 1986.
- [14] Edward A. Snow. *Automation of Module Set Independent Register-Transfer Level Design*. PhD thesis, Carnegie-Mellon University, April 1978.
- [15] Howard Trickey. Flamel: a high-level hardware compiler. *IEEE Transactions on Computer-Aided Design*, CAD-6(2):259–269, March 1987.
- [16] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. MIT Press, 1984.
- [17] Gregory Whitcomb, Brian O'Krafska, and A. Richard Newton. The hardware data-flow representation and synthesis methodology. October 1989. Presented at the 1989 ACM/IEEE Behavioral Synthesis Workshop.
- [18] Donald E. Thomas, Robert L. Blackburn, and Jayanth V. Rajan. Linking the behavioral and structural domains of representation for digital system design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-6(1):103–110, January 1987.
- [19] Srinivas Devadas. Approaches to multi-level sequential logic synthesis. In *Proceedings, 26th Design Automation Conference*, pages 270–276, ACM/IEEE, June 1989.
- [20] Srinivas Devadas, Hi-Keung Ma, A. Richard Newton, and A. Sangiovanni-Vincentelli. MUSTANG: state assignment of finite state machines targeting multilevel logic implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-7(12):1290–1300, December 1988.
- [21] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: a multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-6(6):1062–1081, November 1987.
- [22] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. In J. A. Darringer and F. J. Rammig, editors, *Computer Hardware Description Languages and their Applications*, pages 281–295, Elsevier Science Publishers B. V. (North-Holland), 1990.
- [23] Zvi Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, New York, second edition, 1978.
- [24] Wayne Wolf. An algorithm for nearly-minimal collapsing of finite-state machine networks. February 1990. Submitted to IEEE Transactions on CAD/ICAS.
- [25] Michael Lightner and Wayne Wolf. Experiments in logic optimization. In *Proceedings, ICCAD-88*, pages 286–289, ACM/IEEE, 1988.
- [26] Michael C. McFarland, S. J. Reevaluating the design space for register-transfer hardware synthesis. In *Proceedings, ICCAD-87*, pages 262–265, IEEE Computer Society Press, 1987.